Using MMX™ Instructions to Implement a Video Loop Filter

Information for Developers and ISVs

From Intel® Developer Services
www.intel.com/IDS

March 1996

Using MMX™ Instructions to Implement a Video Loop Filter

March 1996

CONTENTS

1.0. INTRODUCTION

The Intel Architecture (IA) media extensions include single-instruction, multi-data (SIMD) instructions. This application note presents the basics of a loop filter implementation using MMX instructions.

Filtering or smoothing operations are used to reduce noise in imagery that is often characterized by high frequency components. In the loop filter calculation described here, smoothing in YUV space is performed over each frame.

2.0. LOOP FILTER

The 2-D convolution kernel for the loop filter is shown in Figure 1. This 2-D convolution kernel of size 3x3 is equivalent to a 1-D convolution kernel along the rows with coefficients [1 2 1] and a 1-D convolution kernel along the columns with the same coefficients [1 2 1].

Figure 1. 2-D Convolution Kernel

$$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

Notice that the convolution kernel is normalized by the factor 1/16. Normalization is necessary since the sum of all coefficients in the filter must equal one to preserve scaling.

The 2-D loop filter is implemented as two smaller 1-D filters, namely a [1 2 1] filter along the rows ("row_filter") and a [1 2 1] filter along the columns (" col_filter"). Each of these filters is basically an inner product of the data with the [1 2 1] kernel.

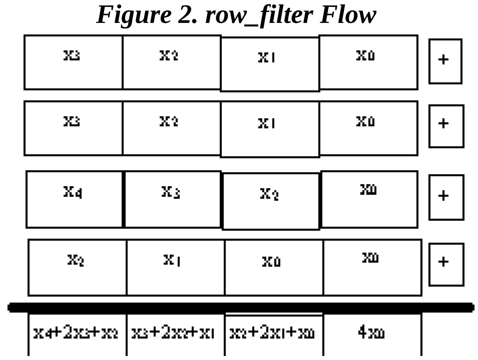
The data is processed in blocks; each block is 8 pixels by 8 pixels in size. Each block passes first through the row_filter and then through the col_filter.

2.1. row_filter Core

Before data passes through the row_filter, it is unpacked from bytes to words for precision. Figure 2 illustrates how the row_filter operates on the lower four words
(0-3). The data element is copied three times. One copy is unchanged, one is shifted left; and one is shifted right. Finally the four resulting data elements are added together. The result is the inner product of the data with the [1 2 1] kernel.

Notice that a boundary condition occurs at the zeroth element that requires special handling. If there were no boundary, the sum for the zeroth element would be x1 + 2x0 + x-1 However, since there is no neighboring data, x_{-1} , we weight the value by a factor of 41 instead. This is achieved by adding a masked out version of $2x_0$ (line 26 of the code, see Example 1).

Figure 2. row_filter Flow



The operation shown in Figure 2 must be repeated for the higher four words (4-7), with similar treatment for the upper boundary condition at the seventh element. Then, the entire process must be repeated for each row of 8 pixels.

Example 1. row_filter Code

```
row_loop:
  1 movq        mm0, [esi]                         ; get a row
  2 pxor        mm7, mm7                  ; clear for unsigned unpacking
  3 movq        mm1, mm0                  ; copy row
  4 psrlq               mm0, 32                              ; align
  5 movq        mm2, mm1                  ; copy row
  6 punpcklbw           mm0, mm7                   ; bytes to word [7 6 5 4]
  7 movq        mm3, mm2                  ; copy row
  8 punpcklbw           mm1, mm7                   ; bytes to word [3 2 1 0]
  9 movq        mm4, mm0                  ; copy half row [7 6 5 4]
 10 psrlq               mm2, 24                              ; align [_ _ _ 7 6 5 4 3]
 11 movq        mm5, mm1                  : copy half row [3 2 1 0]
 12 psrlq               mm3, 8                               ; align [_ 7 6 5 4 3 2 1]
 13 paddw               mm0, mm0                   ; double [7 6 5 4]
 14 punpcklbw           mm2, mm7                   ; bytes to word [6 5 4 3]
 15 paddw               mm1, mm1                   ; double [3 2 1 0]
 16 punpcklbw           mm3, mm7                   ; bytes to word [4 3 2 1]
 17 pand        mm2, DWORD PTR _MASK7              ; make  [_ 5 4 3]
 18 psrlq               mm4, 16                              ; align [_ 7 6 5]
 19 pand        mm3, DWORD PTR _MASK0              ; make  [4 3 2 _]
 20 psllq               mm5, 16                              : align [2 1 0 _]
 21 paddw               mm2, mm4                   ; make [___ 5+7 4+6 3+5]
 22 paddw               mm3, mm5                   ; make [2+4 1+3 0+2]
 23 paddw               mm2, mm0                   ; make [2*7 5+7+2*6 4+6+2*5 3+5+2*4]
 24 pand        mm0, DWORD PTR _NOT_MASK7 ; make [2*7    -       -       -   ]
 25 paddw               mm3, mm1                   ; make [2+4+2*3 1+3+2*2 0+2+2*1 2*0]
 26 pand        mm1, DWORD PTR _NOT_MASK0 ; make [   -       -       -    2*0]
 27 paddw               mm2, mm0                   ; make [4*7 5+7+2*6 4+6+2*5 3+5+2*4]
 28 paddw               mm3, mm1                   ; make [2+4+2*3 1+3+2*2 0+2+2*1 4*0]
 29 movq        _lf_blk[edi], mm3                  ; Store first half of the row
 30 movq        _lf_blk+8[edi], mm2                ; Store second half of the row
 31 add                 edi, 16
 32 add                 esi, 176
 33 dec                 ecx
 34 jnz                 row_loop                   ; Process 8 rows of data
 35 ret
```

The row_filter code is listed in Example 1. Within the loop, one row of pixels is processed. First, the data is unpacked from bytes to words (lines 6 and 8). Register MM0 contains the higher four words; register MM1 contains the lower four words.

Next, the inner product is calculated as follows:

$$x_i = x_{i-1} + 2x_i + x_{i+1}$$

Look at this calculation for the higher four words. Line 13 calculates the values 2xi (stored in MM0). Lines 9 and 18 compute the values xi+1 by copying the data and shifting right (stored in MM4). Lines 5, 10, and 14 compute values xi-1 by copying the data, shifting right, and then unpacking (stored in MM2).

The code handles the boundary condition at the seventh and zeroth elements by preparing registers with doubled boundary values (lines 24 and 26, respectively).

The inner product of the four upper words is formed by adding the three registers together (lines 21 and 23).

Similar calculations are made for the inner products of both the higher and lower halves of the row. Then, the loop is repeated eight times for eight rows of data.

col_filter Core

Figure 3 illustrates how the col_filter performs an inner product of the results of the row_filter with the [1 2 1] kernel. This time, the rows are added together, forming the [1 2 1] results along the columns (i.e., across the rows). Figure 3 shows the flow of the summation across the rows. As before, boundary conditions exist for the first and last rows, so they are handled in a similar fashion as in the case of the row_filter.

Figure 3. col_filter Flow

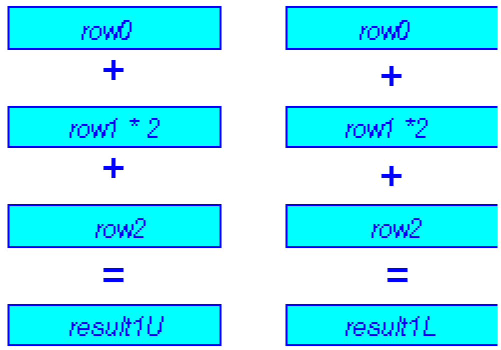
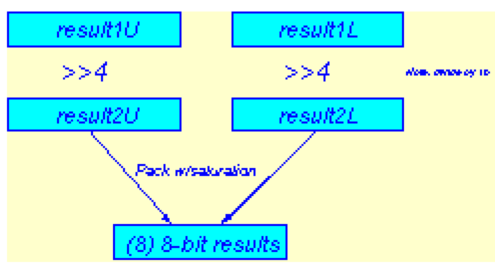


Figure 4 shows how the results are normalized and packed before they are stored in memory. The results are normalized by shifting the result right by 4 places (i.e., dividing by 16). Then the upper and lower results are packed into bytes (with saturation). Packing is necessary because the resulting data elements must be the same size as the input, even though the intermediate calculations were done at twice the precision. As before, boundary conditions are handled separately for the first and last rows in the filter.

Figure 4. Normalizing and Packing the Results



Example 2 lists a small segment of the col_filter code (the loop is completely unrolled for the col_filter). In this code, registers MM2 and MM3 accumulate the rows (lines 4 and 6-10). Lines 11 and 12 normalize the results by shifting right 4 places. Finally, line 9 packs the words back into bytes. Since the col_filter loop has been unrolled, code from different iterations overlaps due to scheduling.

Example 2. col_filter Code

```
1  movq          mm6, _lf_blk+48          ; load row i+1 into mm6
2  packuswb      mm0, mm1         ; row 1 calculation
3  movq          mm7, _lf_blk+56          ; load row i+2 into mm7 (row 3 iter)
4  paddw mm2, mm4          ; accumulate row i-1 + row i
5  movq          frame_y+176[edi], mm0    ; Store results in row 1
6  paddw mm3, mm5
7  paddw mm2, mm4          ; add row i again
8  paddw mm3, mm5
9  paddw mm2, mm6          ; add row i+1
10 paddw mm3, mm7
11 psrlw mm2, 4                   ; normalize result
12 psrlw mm3, 4
```

```
movq          mm0, _lf_blk+64
packuswb      mm2, mm3         ; pack results back to bytes
movq          mm1, _lf_blk+72
paddw mm4, mm6
mov           frame_y+352[edi], mm2     ; Store results in row 2
```

3.0. PERFORMANCE GAINS

Table1 indicates that the video loop filter coded with MMX instructions performed 1.9X faster than the scalar version of the filter. The data represents the simulation of scalar code and MMX code on a Pentium® processor. The simulation processed 30 blocks of data; each block was 8 pixels by 8 pixels.

The performance increase is due primarily to the ability to exploit the parallelism within the filter. That is, the process is separated into two 1-D filters that are performed in parallel using paddw (with only 1 clock latency for four additions, in parallel). First, the calculation is performed along each row, in conjunction with shifts to form a [1 2 1] filter. Then the calculation is performed along the columns (i.e. across the rows) to form a [1 2 1] filter in the orthogonal direction.

Table . TT_PerfGains Performance Gains

	Scalar Code	**MMX™ Code**
Instructions	20315	11675
Cycles	20003	10549
CPI	0.98	0.90

4.0. LOOP FILTER FUNCTION CODE LISTING

```
        .486P
 ASSUME ds:FLAT, cs:FLAT, ss:FLAT
_TEXT SEGMENT DWORD PUBLIC USE32 'CODE'
_TEXT ENDS
_DATA SEGMENT PARA  PUBLIC USE32 'DATA'
        ALIGN4
_DATA ENDS
_DATA SEGMENT PARA  PUBLIC USE32 'DATA'
        ALIGN   16
_zero_quad       db 0, 0, 0, 0, 0, 0, 0, 0
_MASK0   db 0, 0, 0ffh, 0ffh, 0ffh, 0ffh, 0ffh, 0ffh
_MASK7   db 0ffh, 0ffh, 0ffh, 0ffh, 0ffh, 0ffh, 0, 0
_NOT_MASK0       db 0ffh, 0ffh, 0, 0, 0, 0, 0, 0
_NOT_MASK7       db 0, 0, 0, 0, 0, 0, 0ffh, 0ffh
EXTRN    _frame_y:DWORD
EXTRN    _lf_blk:DWORD
_DATA ENDS
_TEXT SEGMENT DWORD PUBLIC USE32 'CODE'
row_filter Proc C Public uses esi edi ecx, array_offset:DWORD
;Row loop of 121 Filter */
;mem array_offset
  mov            esi, array_offset
  xor            edi, edi
  mov            ecx, 8
  lea            esi, _frame_y[esi]
row_loop:
  movq           mm0, [esi]                         ; get a row
  pxor           mm7, mm7
  movq           mm1, mm0                   ; copy row
  psrlq          mm0, 32                            ; align
  movq           mm2, mm1                   ; copy row
  punpcklbw      mm0, mm7                   ; bytes to word [7 6 5 4]
  movq           mm3, mm2                   ; copy row
  punpcklbw      mm1, mm7                   ; bytes to word [3 2 1 0]
  movq           mm4, mm0                   ; copy half row [7 6 5 4]
  psrlq          mm2, 24                            ; align [_ _ _ 7 6 5 4 3]
  movq           mm5, mm1                   ; copy half row [3 2 1 0]
  psrlq          mm3, 8                             ; align
  paddw          mm0, mm0                   ; double [7 6 5 4]
  punpcklbw      mm2, mm7                   ; bytes to word [6 5 4 3]
  paddw          mm1, mm1                   ; double [3 2 1 0]
  punpcklbw      mm3, mm7                   ; bytes to word [4 3 2 1]
  pand           mm2, DWORD PTR _MASK7              ; make  [_ 5 4 3]
  psrlq          mm4, 16                            ; align [_ 7 6 5]
  pand           mm3, DWORD PTR _MASK0              ; make  [4 3 2 _]
  psllq          mm5, 16                            ; align [2 1 0 _]
  paddw          mm2, mm4                   ; make [___ 5+7 4+6 3+5]
  paddw          mm3, mm5                   ; make [2+4 1+3 0+2]
  paddw          mm2, mm0                   ; make [2*7 5+7+2*6 4+6+2*5 3+5+2*4]
  pand           mm0, DWORD PTR _NOT_MASK7 ; make [2*7     -       -       -   ]
  paddw          mm3, mm1                   ; make [2+4+2*3 1+3+2*2 0+2+2*1 2*0]
  pand           mm1, DWORD PTR _NOT_MASK0 ; make [   -       -       -    2*0]
  paddw          mm2, mm0                   ; make [4*7 5+7+2*6 4+6+2*5 3+5+2*4]
  paddw          mm3, mm1                   ; make [2+4+2*3 1+3+2*2 0+2+2*1 4*0]
  movq           _lf_blk[edi], mm3                  ; Store first half of the row
  movq           _lf_blk+8[edi], mm2                ; Store second half of the row
  add            edi, 16
  add            esi, 176
  dec            ecx
```

```
  jnz           row_loop                 ; Process 8 rows of data
  ret
row_filter EndP
col_filter Proc C Public uses edi, array_offset:DWORD
;121 Filter kernel for column section
;mem array_offset
  mov           edi, array_offset
  movq          mm0, _lf_blk
  movq          mm1, _lf_blk+8
  psrlw         mm0, 2
  movq          mm2, _lf_blk+16
  psrlw         mm1, 2
  movq          mm3, _lf_blk+24
  movq          mm7, mm0
  movq          mm4, _lf_blk+32
  packuswb      mm7, mm1
  movq          mm5, _lf_blk+40
  psllw         mm0, 2
  movq          _frame_y[edi], mm7               ; Store results in row 0
  psllw         mm1, 2
  paddw         mm0, mm2
  paddw         mm1, mm3
  paddw         mm0, mm2
  paddw         mm1, mm3
  paddw         mm0, mm4
  paddw         mm1, mm5
  psrlw         mm0, 4
  psrlw         mm1, 4
  movq          mm6, _lf_blk+48
  packuswb      mm0, mm1
  movq          mm7, _lf_blk+56
  paddw         mm2, mm4
  movq          _frame_y+176[edi], mm0   ; Store results in row 1
  paddw         mm3, mm5
  paddw         mm2, mm4
  paddw         mm3, mm5
  paddw         mm2, mm6
  paddw         mm3, mm7
  psrlw         mm2, 4
  psrlw         mm3, 4
  movq          mm0, _lf_blk+64
  packuswb      mm2, mm3
  movq          mm1, _lf_blk+72
  paddw         mm4, mm6
  movq          _frame_y+352[edi], mm2   ; Store results in row 2
  paddw         mm5, mm7
  paddw         mm4, mm6
  paddw         mm5, mm7
  paddw         mm4, mm0
  paddw         mm5, mm1
  psrlw         mm4, 4
  psrlw         mm5, 4
  movq          mm2, _lf_blk+80
  packuswb      mm4, mm5
  movq          mm3, _lf_blk+88
  paddw         mm6, mm0
  movq          _frame_y+528[edi], mm4   ; Store results in row 3
  paddw         mm7, mm1
  paddw         mm6, mm0
  paddw         mm7, mm1
  paddw         mm6, mm2
  paddw         mm7, mm3
  psrlw         mm6, 4
  psrlw         mm7, 4
```

```
  movq          mm4, _lf_blk+96
  packuswb      mm6, mm7
  movq          mm5, _lf_blk+104
  paddw         mm0, mm2
  movq          _frame_y+704[edi], mm6    ; Store results in row 4
  paddw         mm1, mm3
  paddw         mm0, mm2
  paddw         mm1, mm3
  paddw         mm0, mm4
  paddw         mm1, mm5
  psrlw         mm0, 4
  psrlw         mm1, 4
  movq          mm6, _lf_blk+112
  packuswb      mm0, mm1
  movq          mm7, _lf_blk+120
  paddw         mm2, mm4
  movq          _frame_y+880[edi], mm0    ; Store results in row 5
  paddw         mm3, mm5
  paddw         mm2, mm4
  paddw         mm3, mm5
  paddw         mm2, mm6
  paddw         mm3, mm7
  psrlw         mm2, 4
  psrlw         mm3, 4
  packuswb      mm2, mm3
  movq          _frame_y+1056[edi], mm2   ; Store results in row 6
  psrlw         mm6, 2
  psrlw         mm7, 2
  packuswb      mm6, mm7
  movq          _frame_y+1232[edi], mm6   ; Store results in row 7
  ret
col_filter EndP
_TEXT ENDS
END
```